UNDERSEA: An Exemplar for Engineering Self-Adaptive Unmanned Underwater Vehicles

Simos Gerasimou, Radu Calinescu
Department of Computer Science
University of York, UK
{simos.gerasimou, radu.calinescu}@york.ac.uk

Stepan Shevtsov* and Danny Weyns*+
Department of Computer Science
*Linnaeus University, Sweden
+KU Leuven, Belgium
stepan.shevtsov@lnu.se; danny.weyns@kuleuven.be

***Abstract*—Recent advances in embedded systems and underwater communications raised the autonomy levels in unmanned underwater vehicles (UUVs) from human-driven and scripted to adaptive and self-managing. UUVs can execute longer and more challenging missions, and include functionality that enables adaptation to unexpected oceanic or vehicle changes. As such, the simulated UUV exemplar *UNDERSEA* introduced in our paper facilitates the development, evaluation and comparison of self-adaptation solutions in a new and important application domain. UNDERSEA comes with predefined oceanic surveillance UUV missions, adaptation scenarios, and a reference controller implementation, all of which can easily be extended or replaced.**

***Index Terms*—unmanned underwater vehicle exemplar; self-adaptive embedded systems; oceanic surveillance**

I. Introduction

Computer science researchers have long advocated the use of exemplars and benchmarks as a way to promote good practice and encourage high-quality research [1], [2]. These artifacts do not only enable replication and extension of published research, but they also support easy exploration, rapid prototyping and rigorous evaluation of new techniques and approaches. Illustrative examples include well-known benchmark suites for computer vision [3], [4] and probabilistic model checking [5], test problems for multi-objective optimisation [6], and machine learning repositories for data mining, classification and regression. [1]

Despite the advances in the area of adaptive and self-managing systems over the past fifteen years, the use of exemplars for standardising research conducted in the area is still immature [7]. The successful Znn.com news service exemplar [8] paved the way to change this. An ongoing effort is in progress to providing exemplars from different application domains, and several have been proposed recently. For instance, the *TAS* [9] and *Hogna* [10] exemplars enable research in the domains of service-based systems and cloud computing, respectively. *Feed me, Feed me* [11] provides the means to explore and analyse the requirements and characteristics of modern Internet-of-Things-based self-adaptive systems.

In this paper, we propose UNDERSEA, an exemplar for conducting research on self-adaptive systems in the domain of unmanned underwater vehicles (UUVs). These vehicles are used in a wide range of oceanographic and military tasks, including oceanic surveillance (e.g., to monitor pollution levels and ecosystems), undersea mapping, and mine detection [12]. Increasingly UUVs have more powerful on-board processing components, are equipped with sensors with improved capabilities in terms of cost, size, and power consumption, and employ better underwater communication mechanisms. These advances made longer and more complex UUV missions possible. As a result, there is a growing need for fully autonomous and self-adaptive UUVs capable of completing successfully long missions, and self-adapting in response to the unexpected ocean environment and vehicle changes.

The UUV case study was originally introduced in [13] and has already been used in the evaluation of self-adaptation solutions [14], [15], albeit based on ad-hoc implementations and scenarios that make the comparison of these solutions with other solutions difficult. To address these limitations, we developed UNDERSEA, which is packaged with predefined scenarios for evaluating self-adaptation solutions.

UNDERSEA is a simulated UUV exemplar built on top of the open-source middleware MOOS-IvP[2], a widely used platform for the implementation of autonomous applications on UUVs [16]. As such, the code of self-adaptation solutions developed with UNDERSEA can be directly used on actual UUVs that run MOOS-IvP. Like many approaches to engineering self-adaptive systems (e.g., [17], [18], [19]), UNDERSEA adopts the conventional MAPE-K control loop [20] and comprises a simulated managed system (UUV) and its controller. However, other types of control loops can be plugged in, for example, controllers based on principles from control theory. The exemplar is available preinstalled on an easy-to-use virtual machine and supports a range of UUV missions and adaptation scenarios. Also, new missions and adaptation scenarios can be defined with limited effort. Finally, the real-time update of the UUV simulator combined with the decoupling of the observation side (*shoreside*) from the operation side (UUV) provides a realistic UUV mission in which the effect of adaptation decisions is timely visualised on the shoreside.

The rest of the paper is structured as follows. Section II overviews UNDERSEA and its supported missions. Section III presents the architecture of the exemplar. Section IV describes

[1] e.g., archive.ics.uci.edu/ml, kdnuggets.com/datasets/index.html

[2] Mission Oriented Operating Suite - Interval Programming

the process of engineering a self-adaptation solution with UNDERSEA, and illustrates it for a solution whose MAPE controller uses runtime probabilistic model checking. Finally, Section V concludes the paper with a short summary.

II. UNDERSEA Overview

UNDERSEA simulates a UUV deployed to carry out an environmental surveillance/data gathering mission. The UUV is equipped with $n \geq 1$ on-board sensors that can measure a parameter of the ocean environment such as water current, salinity or temperature. The n sensors can be switched on and off individually (e.g., to save battery power when not required), but these operations consume an amount of energy. The energy consumed to switch on sensor i is denoted e_i^{on}, and the energy to switch off sensor i is e_i^{off}, for $1 \leq i \leq n$. When sensor i is switched on, it takes measurements of the oceanic parameter under study with a variable rate r_i. Finally, the probability that a measurement is sufficiently accurate for the purpose of the mission depends on the UUV speed sp, and is given by p_i.[3]

The UUV needs to be augmented with a controller that dynamically adjusts:

(a) the UUV speed sp

(b) the sensor configuration $x_1, x_2, \ldots, x_n$ (where $x_i = 1$ if the i-th sensor is switched on and $x_i = 0$ otherwise)

so that the self-adaptive system obtained through the integration of the UUV and the controller handles the generic adaptation scenarios from Table I.

Within these scenarios, we propose the evaluation and comparison of different self-adaptation solutions based on:

- Ability to resume compliance with requirements after the unpredictable events and changes;
- Utility achieved by the UUV;
- Distance covered by the UUV during the mission;
- Computational overhead to run the controller.

III. UNDERSEA Architecture

A. The MOOS-IvP Middleware

UNDERSEA is developed using MOOS-IvP, an established open-source middleware for engineering autonomous applications on unmanned marine vehicles [16]. When used for the execution of oceanic missions, MOOS-IvP is deployed on the payload computer of an autonomous vehicle, so as to decouple vehicle autonomy from the navigation and control system running on the main vehicle computer [16].

A MOOS-IvP-based system is a community of independent applications running in parallel. These applications communicate through a MOOS database (MOOSDB) using a publish-subscribe architecture (Fig. 1). To this end, applications can publish messages in the form of key–value pairs with agreed frequencies. These messages can provide information about the vehicle components monitored by an application, e.g., after an on-board sensor from the UNDERSEA exemplar performs a reading, it publishes a message which summarises the performed action. Any interested "listener" applications can subscribe to these messages (using the appropriate keys) and act upon receiving an update, e.g., the UUV middleware from Fig. 1 subscribes to the messages transmitted by its on-board sensors, and when a new message arrives it adjusts its estimate of the average rate of the relevant sensor.

Vehicle autonomy in MOOS-IvP is guided by a collection of *behaviours*, i.e., combinations of boolean logic constraints and piecewise-linear utility functions parameterised, for instance, by parameters of the navigation and control system such as $heading$, $speed$ or $depth$. User-defined MOOS applications can propose behaviours and thus affect vehicle autonomy. During a mission, the IvP Helm, the decision-making component of the platform, periodically collects the proposed behaviours. When multiple behaviours are active, this component carries out Interval Programming (IvP) multi-objective optimisation to establish the optimal action, i.e., an optimal point in the decision space defined by the constraints and utility functions. This optimal action is expressed as a set of key–value pairs and is published to the MOOSDB so that other interested applications can receive it.

B. UNDERSEA Realisation with MOOS-IvP

UNDERSEA (Fig. 2) comprises a *shoreside*, a *UUV controller* and a *managed UUV system*. These components run independently and communicate through a client-server architecture. The configuration parameters for the three components are specified in a *mission file*, expressed in an easy-to-use domain-specific language (DSL) (Fig. 3). The *Mission parser*, built using the Antlr parser generator (http://www.antlr.org) checks that the mission file contains all the required settings, and extracts the configuration parameters for the three UNDERSEA components. For example, a new sensor can be included by adding a new SENSOR block (lines 12–18 in Fig. 3); if another sensor with the same name exists or a required setting is omitted, the parser will throw an exception.

The *Managed UUV system* comprises the *UUV middleware*, a *Sensor* application that we specifically built for UNDERSEA, and other standard applications required by MOOS-IvP to run the system (e.g., IvP Helm, MOOSDB) that cannot be adapted by the UUV controller (not shown in Fig. 2). All configuration information for the UUV is provided in the

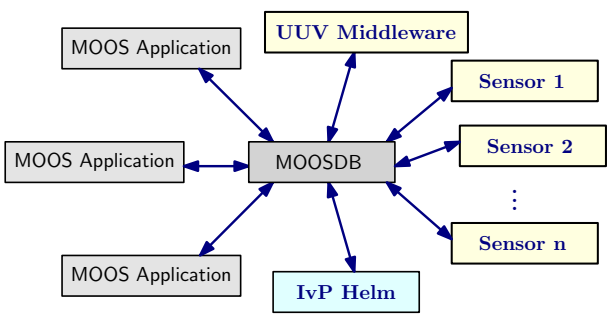

Fig. 1. MOOS-IvP architecture, adapted from [16]

[3]This information can be extracted from the technical specification of sensors; for example, see http://www.ashtead-technology.com/rental-equipment/teledyne-rdi-600khz-navigator/

TABLE I
GENERIC ADAPTATION SCENARIOS FOR OCEANIC SURVEILLANCE UUV SYSTEMS

Scenario	Type of uncertainty [21]	Type of adaptation [13], [14]	Type of requirements
S1	Unpredictable environment: sensor degradation	Switch on additional/alternative sensor(s); Switch off degraded sensor	QoS: Throughput, energy usage
S2	Unpredictable environment: sensor failure	Switch on functionally-equivalent sensor(s)	QoS: Performance, reliability, utility
S3	Unpredictable environment: sensor repair	Switch on repaired sensor; Switch off expensive sensor(s)	QoS: Performance, utility
S4	Changing requirements: new goals	Change set of active sensors	QoS: Performance, cost

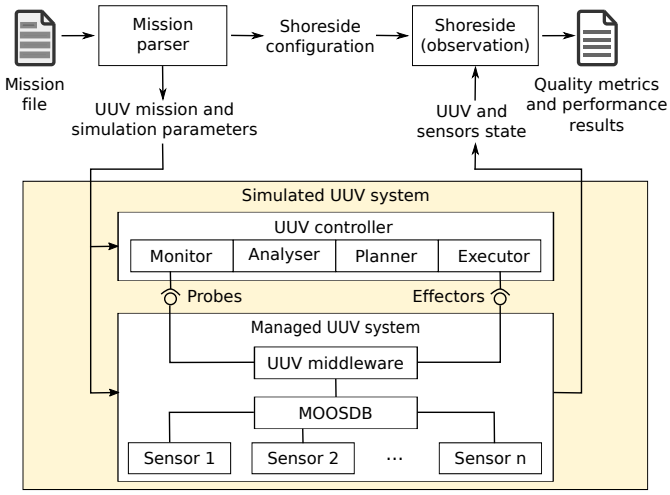

Fig. 2. High-level UNDERSEA architecture.

```
1  simulation time = 2000
2  time window = 5
3  host = localhost
4  port = 9999
5
6  UUV {
7    name = Nautilus
8    port = 8888
9    speed = 0:5:20
10 }
11
12 SENSOR {
13   name = Sensor1
14   rate = 5
15   reliability = 0.9
16   change = 50:100:3
17   change = 150:200:4.5
18 }
```

Fig. 3. Fragment of a mission file specified in the UNDERSEA DSL.

mission file, including the hostname and port used by the on-board sensors and the UUV middleware to communicate with MOOSDB and exchange messages (lines 3 and 4 in Fig. 3). The UUV configuration settings include the UUV name, the port through which the managed UUV system communicates with the controller, and the range of possible speed values in the format $MinSpeed:MaxSpeed:Intervals$ (lines 6–10). For example, the command on line 9 specifies that the UUV speed $sp \in \{0, 0.25, 0.50, ..., 4.75, 5.00\}$. For each on-board sensor, a relevant configuration block is defined (lines 12–18), and MOOS-IvP launches a new Sensor instance with the given configuration. Measurement-rate change patterns for a sensor are specified using the *change* command (e.g., line 16) in the format $StartTime:EndTime:NewRate$. For instance, the command on line 16 means that during the simulated time interval $[50, 100]$ Sensor1 will operate with a rate of 3Hz.

The *UUV controller* is where new self-adaptation solutions (e.g., new adaptation algorithms, optimisation strategies, or learning techniques) can be developed and integrated for evaluation. To facilitate the implementation of new solutions, the UNDERSEA distribution provides abstract Java classes that delineate the functionality of each MAPE loop element. Fig. 4, for instance, shows the abstract *Executor* class. Developing a new controller requires simply the extension of these abstract classes, to specialise their unimplemented methods, and to inform the *UUV controller* component about the new classes.

At regular time intervals (defined by *time window* in Fig. 3 – lines 2), the UUV controller uses the *Probes* to retrieve the current system state from the *UUV middleware*, i.e., the average rate of the on-board sensors and the UUV speed. The controller then runs a MAPE loop, selects the desired vehicle speed and sensors configuration, and communicates its decision to the managed UUV system through *Effectors*. The UUV middleware receives this decision and enforces the IvP Helm to adapt the behaviour of the UUV system by realising the new configuration.

Extending the set of data associated with the managed

```
1  package controller;
2
3  public abstract class Executor {
4    /** Create a new executor instance*/
5    public Executor() {...}
6
7    /** Where the actual work is done*/
8    public abstract void run();
9
10   /** Get the command in the form:
11   speed=value, sensor_name=x, sensor_name=x,... */
12   public abstract String getCommand();
13   ...
14 }
15
```

Fig. 4. Executor abstract class.

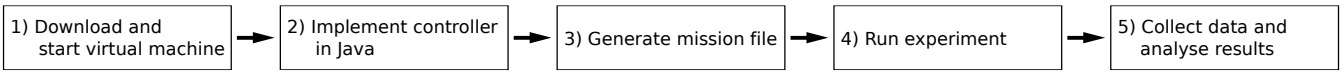

Fig. 5. Workflow for engineering and evaluating self-adaptive solutions with UNDERSEA.

system state (e.g., with depth or coast proximity data) requires (i) enhancing the sensor applications currently available within the managed UUV system (i.e., Sensor 1, ..., Sensor n) or developing new data specific applications (e.g., Depth_Sensor); (ii) making the UUV middleware aware of the new data; and (iii) modifying the Probes to enable retrieving the new data from the UUV middleware. Interested researchers can follow the guide available at [22]. These extensions enable exploring adaptation scenarios beyond those reported in Table I and facilitate the design and evaluation of controllers that can cope with a wider range of uncertainty and requirement types.

In line with the MOOS-IvP design principles, we decoupled the shoreside "observation room" and the (simulated) UUV system. We made this decision to simplify the implementation of multi-UUV missions in future versions of the exemplar. Such multi-UUV missions could be used to assess simultaneously the performance of different controllers running on the same platform. Alternatively, we could deploy a multi-UUV mission (with all UUVs running the same controller) on separate platforms (e.g., on a standard laptop and a Raspberry Pi) and compare the controller performance on these machines.

Once simulation finishes (given by *simulation time* in Fig. 3), UNDERSEA exports a set of files for analysing the controller performance. This set includes a summary of quality metrics for the executed mission, a log file with data about the state of the managed system and the configuration selected by the controller, and a synopsis of CPU and memory usage.

IV. Using UNDERSEA

A. Overview

Researchers using UNDERSEA need to carry out the activities shown in Fig. 5. All the dependencies in the system are preinstalled and the system is fully configured within a virtual machine that researchers can obtain in Step 1 from our project websitehttp://www-users.cs.york.ac.uk/simos/UNDERSEA. [4]

Step 2 involves the implementation of the controller by specialising the abstract Java classes of the UUV controller. One of our existing controllers can be selected or a dedicated controller can be implemented. We use the build automation tool Apache Maven (maven.apache.org) to manage the controller and its dependencies.

Step 3 involves defining the mission file with the structure described in the previous section. For convenience, the virtual machine includes a set of predefined missions. Executing an UNDERSEA build script causes the invocation of the Mission parser (Fig. 2) to verify and extract the parameters for the shoreside, the controller and the managed UUV system.

[4]The project website includes also a step-by-step guide for installing UNDERSEA locally.

Step 4 involves starting the experiment by using an UNDERSEA launch script. The UUV simulator console is automatically displayed, presenting mission-related information (e.g., active sensors and UUV speed) as shown in Fig. 7. This can help to identify and correct early any controller or mission configuration problems.

Finally, Step 5 involves collecting the mission data, and analysing the mission and the performance of the controller.

B. Case Study

We illustrate the five-step engineering process described above and the UNDERSEA capabilities using a case study adapted from our previous work [13]. The UUV used in this case study is travelling with speed $sp \in [0, 5\text{m/s}]$ and is equipped with three on-board sensors that operate with nominal reading rates $r_1 = 5\text{Hz}$ and $r_2 = r_3 = 4\text{Hz}$, and consume energy per reading $e_1 = 3\text{J}$, $e_2 = 2.4\text{J}$ and $e_3 = 2.1\text{J}$. The amounts of energy consumed to switch the sensors on and off are $e_1^{\text{on}} = 10\text{J}$, $e_2^{\text{on}} = 8\text{J}$, $e_3^{on} = 5\text{J}$ and $e_1^{\text{off}} = 2\text{J}$, $e_2^{\text{off}} = 1.5\text{J}$, $e_3^{\text{off}} = 1\text{J}$, respectively. A reading is accurate with probability $p_i = 1 - \alpha_i sp$, $1 \leq i \leq 3$, where $\alpha_i \in (0, 0.15)$ is a sensor-dependent accuracy factor. Finally, the UUV should self-adapt so that the following QoS requirements are satisfied:

R1 (throughput): The UUV should take at least 20 readings of sufficient accuracy per 10 metres of mission distance.

R2 (resource usage): The energy consumption of the sensors should not exceed 120 Joules per 10 surveyed metres.

R3 (cost): If requirements R1 and R2 are satisfied by multiple configurations, the UUV should use one of these configurations that minimises the cost function

$$cost = w_1 E + w_2 sp^{-1},$$

where E is the energy used by the sensors per 10m travelled by the UUV, and the weights $w_1, w_2 > 0$ express the desired trade-off between carrying out the mission with reduced battery usage and completing the mission faster.

We assume that **Step 1** of the UNDERSEA engineering process from Fig. 5 is completed, and present Steps 2–4 below.

Step 2: Implementing the UUV controller
The proposed controller employs probabilistic model checking (PMC) at runtime [23], [24], [25] to assess UUV compliance with QoS requirements R1–R3 and to perform runtime reconfigurations when needed. To this end, *MonitorPMC*[5] inspects the current sensor rates and determines whether the change specified by this data should be analysed — the controller can

[5]We use the suffix 'PMC' for controller elements implemented using probabilistic model checking at runtime, e.g., MonitorPMC and AnalyserPMC.

```
1  package controllerPMC;
2
3  public class AnalyserPMC extends Analyser{
4
5    /** PRISM API instance (developed in our previous work)*/
6    PrismAPI prism;
7
8    /** Stochastic model and properties files*/
9    String modelFile      = "models/uuv/uuv.sm";
10   String propertiesFile  = "models/uuv/uuv.csl";
11
12   /** Create a new AnalyserPMC instance*/
13   public AnalyserPMC() {
14     /** Instantiate PRISM and assign the properties file*/
15     prism = new PrismAPI(propertiesFile);
16     ...
17   }
18
19   /** Run the analyser (using probabilistic model checking)*/
20   public void run(){
21     /** Get current sensors rate*/
22     double rates[] = Knowledge.getSensorsRates();
23
24     /** For all UUV system configurations*/
25     for (int config=0; config<UUV_CONFIGS; config++){
26       //1) Instantiate parametric stochastic model
27       String model = instantiateStochasticModel(config, rates);
28
29       //2) load the model to PRISM
30       prism.loadModel(model);
31
32       //3) run PRISM
33       List<Double> propertiesResults  = prism.runPrism();
34
35       //4) save configuration results to Knowledge
36       Knowledge.updateResult(config, propertiesResults);
37     }
38   }
39   ...
40 }
41
```

Fig. 6. Excerpt of Analyser class (AnalyserPMC) that uses probabilistic model checking at runtime.

terminate early if there is no (significant) change in the UUV state since its previous invocation.

AnalyserPMC uses the probabilistic model checker PRISM [26] programatically to verify stochastic models of the UUV parametrised with the possible speed and sensor configurations. Fig. 6 shows an excerpt of AnalyserPMC in which we first instantiate the PRISM API and specify the stochastic model and properties files (lines 5–17). When AnalyserPMC runs, it extracts the current sensor rates, instantiates the stochastic model, loads the model into PRISM, invokes the model checker, and finally stores the verification results to *Knowledge* (lines 19–38) for later use. No further implementation effort is required to complete AnalyserPMC.

PlannerPMC examines the verification results, and chooses the vehicle speed and sensor configuration that satisfy the throughput and resource usage requirements R1 and R2, and minimises the cost (cf. requirement R3). Next, *PlannerPMC* assembles a *reconfiguration plan* specifying which sensors need to be switched on or off, and in which order, so that the selected configuration is achieved.

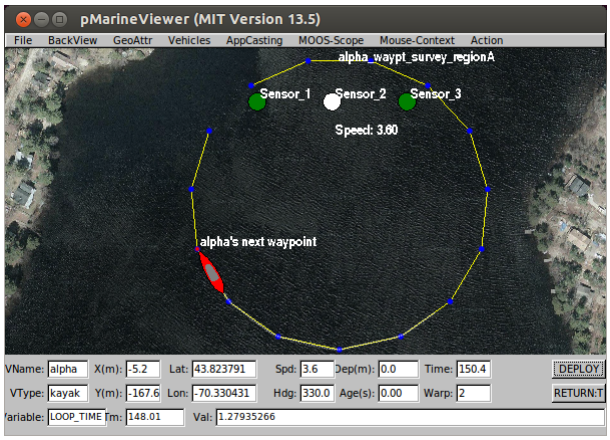

Fig. 7. Self-adaptive UUV simulator

ExecutorPMC uses the reconfiguration plan to synthesise the sequence of commands that will be transmitted by the Effectors to the managed UUV system, ensuring the implementation of the plan.

A fully-working PMC-based controller is available in UNDERSEA distribution for further experimentation. Note that UNDERSEA does not require specialising all MAPE classes, especially if they are not useful for a new controller. UNDERSEA comes with default classes that could be used to complete the controller in such self-adaptation solutions.

Step 3: Specifying the mission file
The mission file for our case study is the one presented in Fig. 3, with two additional 'SENSOR' configuration blocks (lines 12–18) that we did not include in the paper due to space constraints. The full mission file for the case study is available as part of the UNDERSEA distribution. Running the UNDERSEA build script validates the mission file and extracts the required parameters for the shoreside, controller and managed system UNDERSEA components (into separate configuration files).

Step 4: Running the experiment
We start the experiment by executing the UNDERSEA launch script. While UNDERSEA executes the mission (Step 3) and our PMC-based controller (Step 2) drives the UUV adaptation, we receive a summary of the mission evolution in the UNDERSEA console. For instance, Fig. 7 shows a screenshot of our three-sensor mission at a time moment when sensors 1 and 3 are switched on (i.e., $x_1 = x_3 = 1$), sensor 2 is switched off (i.e., $x_2 = 0$), and the UUV speed is $sp = 3.6$m/s.

Step 5: Collecting the data and analysing the results
When UNDERSEA terminates its execution, we obtain a set of experiment-related data. The analysis of this data provides useful insights regarding the CPU and memory overheads of the controller, and its ability to respond adequately to changes affecting the UUV system. For instance, Fig. 9 shows the changes in sensor rates and the new UUV configurations

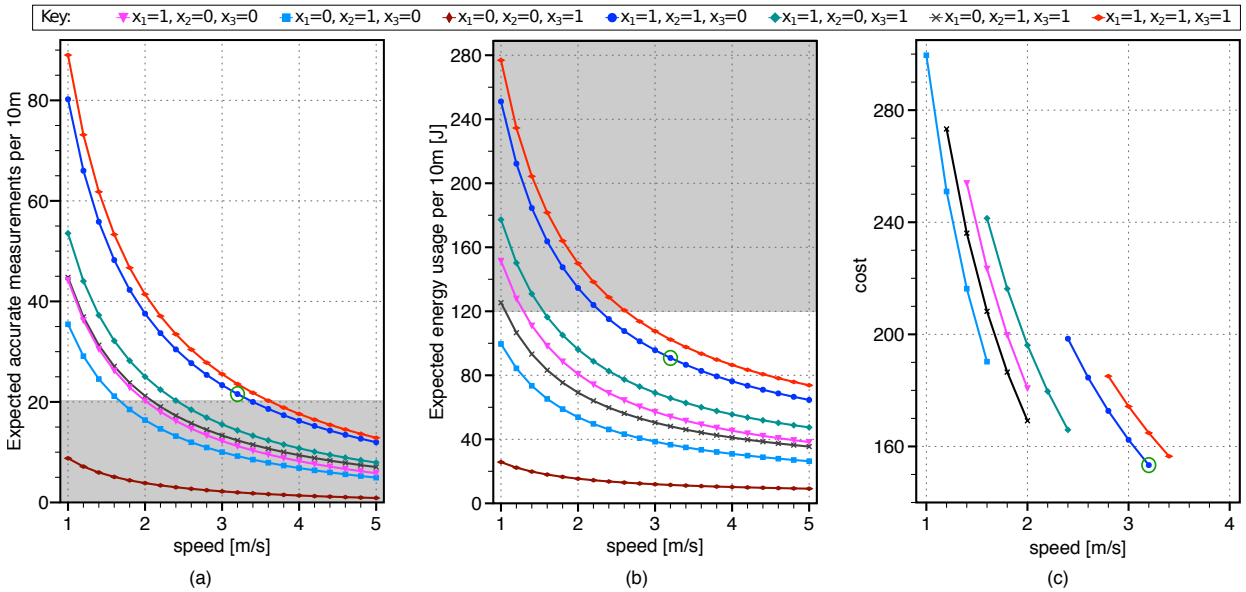

Fig. 8. Verification results for requirement (a) R1, (b) R2, and (c) cost of the feasible configurations. The best configuration (circled) corresponds to $x_1 = x_2 = 1, x_3 = 0$ (i.e. UUV using only its first two sensors) and $sp = 3.2m/s$, and the shaded regions correspond to requirement violations.

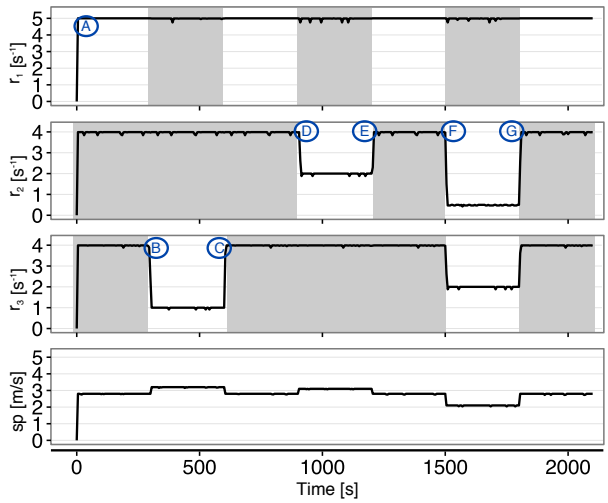

Fig. 9. Change scenarios for the self-adaptive UUV system over 2100 seconds of simulated time. Extended shaded regions indicate the sensors switched on at each point in time.

selected by the PMC-based controller. The labels A–G from Fig. 9 are associated with the following adaptation decisions:

A) The UUV starts with the initial state and configuration;

B) Sensor 3 experiences service degradation ($r_3^{\text{new}} = 1$Hz), so the higher-rate but less energy efficient sensor 1 is switched on (allowing a slight increase in speed to $sp = 3.2$m/s) and sensor 3 is switched off;

C) Sensor 3 recovers and the initial configuration is resumed;

D) Sensor 2 experiences a degradation, and is replaced by sensor 1, with the speed increased to $sp = 3.1$m/s;

E) Sensor 2 recovers and the initial configuration is resumed;

F) Both sensor 2 and sensor 3 experience degradations, so sensor 1 alone is used, with the UUV travelling at a lower speed $sp = 2.1$m/s;

G) Sensors 2 and 3 resume operation at nominal rates and the initial UUV configuration is reinstated.

Finally, using the data available we can perform a deeper analysis of the adaptation decisions made by the PMC-based controller. Fig. 8, for example, depicts the verification results corresponding to label B from Fig. 9 for QoS requirements R1–R3 and the configuration selected by the controller.

V. Conclusion

We introduced UNDERSEA, an exemplar that can help researchers to quickly explore, develop, evaluate and compare new self-adaptation solutions in the unmanned underwater vehicle domain. The exemplar provides a reference implementation that is built on top of the established open-source middleware MOOS-IvP and comes with a set of predefined scenarios for experimentation. We illustrated the use of the exemplar with an example where the controller employs probabilistic model checking at runtime to assess UUV compliance with a set of QoS requirements. The exemplar is extendable and other self-adaptation solutions (e.g., based on control-theory [15] or stochastic search [27]) can be developed easily.

We plan to implement a number of extensions in the near future. First, in the current version, the user gets the data of the simulation, which is the basis for analysing and comparing solutions. In the next release, we plan to provide additional support for automated analysis and comparison of controller solutions. Second, we plan to support research on distributed adaptation by supporting multi-UUV missions. Third, we plan to add complementary scenarios for the evaluation of adaptation solutions, e.g., scenarios focussing on optimising the accuracy of measurements and testing the scalability of adaptation solutions. The exemplar and a video demonstration are available at https://www-users.cs.york.ac.uk/simos/UNDERSEA.

REFERENCES

[1] M. S. Feather, S. Fickas, A. Finkelstein, and A. V. Lamsweerde, "Requirements and specification exemplars," *Automated Software Engineering*, pp. 419–438, 1997.

[2] S. E. Sim, S. Easterbrook, and R. C. Holt, "Using benchmarking to advance research: A challenge to software engineering," in *25th International Conference on Software Engineering (ICSE'03)*, 2003, pp. 74–83.

[3] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the KITTI vision benchmark suite," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'12)*, 2012, pp. 3354–3361.

[4] C. C. Chang and C. J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, 2011.

[5] M. Kwiatkowska, G. Norman, and D. Parker, "The PRISM benchmark suite," in *9th International Conference on Quantitative Evaluation of SysTems (QEST'12)*, 2012, pp. 203–204.

[6] S. Huband, P. Hingston, L. Barone, and L. While, "A review of multiobjective test problems and a scalable test problem toolkit," *Trans. Evol. Comp*, vol. 10, no. 5, pp. 477–506, 2006.

[7] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, B. Schmerl, D. B. Smith, J. P. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, and J. Wuttke, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS, 2013, vol. 7475, pp. 1–32.

[8] S. W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the effectiveness of the rainbow self-adaptive system," in *4th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'09)*, 2009, pp. 132–141.

[9] D. Weyns and R. Calinescu, "Tele assistance: A self-adaptive service-based system examplar," in *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*, 2015, pp. 88–92.

[10] C. Barna, H. Ghanbari, M. Litoiu, and M. Shtern, "Hogna: A platform for self-adaptive applications in cloud environments," in *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*, 2015, pp. 83–87.

[11] A. Bennaceur, C. McCormick, J. G. Galán, C. Perera, A. Smith, A. Zisman, and B. Nuseibeh, "Feed me, feed me: An exemplar for engineering adaptive software," in *11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'16)*, 2016, pp. 89–95.

[12] M. Seto, L. Paull, and S. Saeedi, "Introduction to autonomy for marine robots," in *Marine Robot Autonomy*, 2013, pp. 1–46.

[13] S. Gerasimou, R. Calinescu, and A. Banks, "Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration," in *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*, 2014, pp. 115–124.

[14] R. Calinescu, S. Gerasimou, and A. Banks, "Self-adaptive software with decentralised control loops," in *18th International Conference on Fundamental Approaches to Software Engineering (FASE'15)*, 2015, pp. 235–251.

[15] S. Shevtsov and D. Weyns, "Keep it simplex: Satisfying multiple goals with guarantees in control-based self-adaptive systems," in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*, 2016, pp. 229–241.

[16] M. Benjamin, H. Schmidt, P. Newman, and J. Leonard, "Autonomy for unmanned marine vehicles with MOOS-IvP," in *Marine Robot Autonomy*, 2013, pp. 47–90.

[17] D. Garlan, S.-W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct 2004.

[18] D. Weyns, S. Malek, and J. Andersson, "FORMS: Unifying reference model for formal specification of distributed self-adaptive systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 1, p. 8, 2012.

[19] R. Calinescu, "Implementation of a generic autonomic framework," in *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*, 2008, pp. 124–129.

[20] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[21] A. Ramirez, A. Jensen, and B. Cheng, "A taxonomy of uncertainty for dynamically adaptive systems," in *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12)*, 2012, pp. 99–108.

[22] "An Overview of MOOS-IvP and a Users Guide to the IvP Helm - Release 15.5," http://oceanai.mit.edu/ivpman/pmwiki/pmwiki.php, May 2015.

[23] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Communications of the ACM*, vol. 55, no. 9, pp. 69–77, 2012.

[24] S. Gerasimou, "Runtime quantitative verification of self-adaptive systems," Ph.D. dissertation, University of York, York, UK, 2017.

[25] R. Calinescu, K. Johnson, and Y. Rafiq, "Developing self-verifying service-based systems," in *28th International Conference on Automated Software Engineering (ASE'13)*, 2013, pp. 734–737.

[26] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: verification of probabilistic real-time systems," in *23rd International Conference on Computer Aided Verification (CAV'11)*, 2011, pp. 585–591.

[27] Z. Coker, D. Garlan, and C. Le Goues, "SASS: Self-adaptation using stochastic search," in *10th International Symposium on Software Engineering for Adaptive and Self- Managing Systems (SEAMS'15)*, 2015, pp. 168–174.